List Update with Prediction

Yossi Azar[*1], **Shahar Lewkowicz**[*1], **Varun Suriyanarayana**[*2]

[1]Department of Computer Science, Tel-Aviv University
[2]School of Operations Research and Information Engineering, Cornell University
azar@tauex.tau.ac.il, shaharlewko22@gmail.com, vs478@cornell.edu

Abstract

List Update is a fundamental problem in online algorithms, with a well-known 2-competitive algorithm that moves every requested element to the front. Randomization can slightly improve the competitive ratio to 1.6, but not beyond 1.5. However, practical inputs are not adversarial and one hopes to do better, particularly when additional information from a machine learning oracle is available. With access to predictions, the goal is to incur only a slight overhead compared to the prediction's accuracy, avoiding significant costs in case of substantial deviation.

We propose a $(1+\epsilon)$-smooth randomized algorithm, offering robustness of $O(1/\epsilon^4)$. This guarantees that the algorithm never exceeds a cost greater than $1+\epsilon$ times the prediction cost, while maintaining a bound within $O(1/\epsilon^4)$ of the optimal cost for every possible sequence. In cases where no paid swaps are permitted for the prediction, we can improve robustness to $O(1/\epsilon^2)$ while retaining $1+\epsilon$ smoothness. We complement these findings by demonstrating a lower bound of $\Omega(1/\epsilon)$ on the robustness for deterministic algorithms and $\Omega(\log(1/\epsilon))$ for randomized ones. Finally, the experiments we have made show that our algorithms perform better than the standard competitive algorithms for this problem.

Introduction

The **List Update** problem is a fundamental problem in online algorithms. In this problem, we have an ordered list of elements and receive requests for these elements over time. Upon receiving a request, the algorithm must immediately serve it by accessing the required element. The cost of accessing an element is equal to its position in the list. The algorithm can move this element towards the front of the list for free, and it can also swap any two consecutive elements in the list at a cost of 1. The objective in this problem is to devise an algorithm that minimizes the total cost of accesses and swaps. It is worth noting that this is an online algorithm, meaning it lacks knowledge of future requests and must make decisions solely based on requests that have arrived.

Sleator and Tarjan (Sleator and Tarjan 1985) introduced a surprising deterministic online 2-competitive algorithm called Move to Front (MTF). Upon receiving a request for an element, this algorithm accesses it and then moves it to the front of the list. They also established a simple lower bound of 2 for the competitive ratio of deterministic online algorithms by always requesting the last element in the online algorithm's list. While randomization can improve the competitive ratio, the best-known bound remains approximately 1.6 (Albers, Von Stengel, and Werchner 1995).

While the best competitive ratio bound for the list update problem is well-studied, in practical scenarios, paying twice (or even 50% more than) the optimal solution may not be satisfactory. This is particularly true when additional information is available. Machine learning models, for instance, can analyze historical data to make informed predictions about the order of elements in the list. Designing a predictor based on machine learning can lead to predictions that, if accurate, may only incur a slightly increased cost compared to the optimal value. However, blindly following inaccurate predictions could result in significantly higher costs compared to the optimal solution. Therefore, there is a need to design algorithms whose worst-case performance does not degrade significantly for any possible sequence. Lykouris and Vassilvitskii (Lykouris and Vassilvitskii 2021) introduced the concept of predictions—a machine learned oracle for a model trained on input data—and explored how we can improve the performance of the model using the oracle, measured by smoothness versus robustness as defined later.

Determining the type of prediction to be used

Determining the most suitable prediction oracle is not straightforward. A natural approach would be to obtain predictions of the entire sequence in advance. However, a prediction of this type may not be very accurate as it does not adapt to, learn from, or take advantage of the observed sequence, potentially leading to significant degradation in prediction accuracy. Additionally, the amount of information required for such a prediction could be unreasonably large. It seems more sensible to obtain a prediction for each request, indicating how to handle the accessed element (i.e., where to move it). Moreover, the prediction algorithm may be allowed to suggest exchanging other elements, making the most general prediction the arrangement of the list at any given time. In most cases, the prediction may recommend moving the accessed element towards the front (but

*These authors contributed equally.

not necessarily all the way). In addition, it may also suggest exchanging elements that were not accessed.

Getting a Prediction

There are various ways to get a prediction on the order of the elements in the list, some of which are based on classical machine learning methods of deep networks. Nevertheless, there are simple, straightforward and easy to implement methods that can provide very good estimates on the predicted order of the list.

- It is well known that if the requests are drawn from a fixed distribution and different request are independent, then ordering the requests according to decreasing probabilities is optimal in expectations. Actually the probabilities do not need to be known in advance and the methods of frequency count (i.e. order the elements by the number of accesses in decreasing order) is approaching to decreasing probabilities. Hence, the prediction can be based on frequency count. Note that in this case the prediction only performs free swaps, since when accessing an element its counter is increased, and it can only advance toward the head of the list where the order of the other elements remains fixed.
- One major issue with the above method is that it takes into account the very far past with the same importance as the recent past. Hence, if there is a change in the distribution, it would take quite a bit of time to adapt to that. A possible solution is to base the frequency count on a fixed window of requests. Then far history become irrelevant. However, this method has several difficulties. The first is that you need to keep the sequence in the memory, since you need to drop the oldest request from the count. This requires a large overhead. In addition, since the count of elements also decreased the algorithm may need to use paid swaps. Moreover, this method does not differentiate between very recent history and recent history and in reality such distinction may exist. In order to overcome all these issues we suggest the following method described below.
- Here, we suggest the exponential decay method. The request in time t in the past will be counted in weight γ^t, where $\gamma < 1$ is the decay factor. Then all these weights are normalized to a distribution. An easy way to implement the suggested scheme is to have the vector of probabilities. In each step, the vector is multiplied by γ and the weight of the requested element is increased by $1-\gamma$. Hence, the sum of the weights remains 1. This method seems to have many advantages. Firstly, it counts recent history as more influential than slightly less recent; it can be implemented easily without keeping any history of the sequence; and the prediction only performs free swaps since the order of all elements accept the accessed one remains the same where the accessed one may advance towards the head on the list. We note that the value of γ determines the rate of decay and may depend on the sequence as well.
- Finally, we suggest a variant on the exponential decay method, which is the multiplicative weight method. It is similar to the exponential decay with all its advantages but with a slightly different updated rule. At any time, we have vectors of weights (probabilities) for each element. In each step the weight of the accessed element is increase by a multiplicative factor of $\gamma > 1$ and all the weights are normalized to have unit sum. We note that the normalization is actually not even necessary, since the order of the weighted is the same as the normalized ones.

Measurement and Prediction error

Before presenting our results, we define the notions of robustness and smoothness.

For any algorithm A for the List Update problem and any request sequence σ, $A(\sigma)$ denotes the cost incurred by A when serving σ (expected cost for randomized algorithm). Similarly, $OPT(\sigma)$ denotes the cost of the optimal solution for sequence σ. If σ is clear from context, we may use OPT to denote the optimal algorithm for σ [1].

Definition 0.1. An algorithm ALG is said to be α-robust if $ALG(\sigma) \leq \alpha \cdot OPT(\sigma)$ for all input sequences σ and all possible predictions (i.e., it is α-competitive independent of the prediction). It is said to be β-smooth if $ALG(\sigma) \leq \beta \cdot PRED(\sigma)$ for all input sequences σ and all possible predictions.

We add the common notion in the field of the prediction error, to measure our algorithm, which is simply a different form of the definition above. We define the relative prediction error as:

$$\delta = \frac{PRED(\sigma) - OPT(\sigma)}{OPT(\sigma)}$$

or in other words, $PRED(\sigma) \leq (1+\delta) \cdot OPT(\sigma)$. Clearly $\delta = 0$ is the case that following the prediction yields the optimum cost. The value of δ corresponds to how much the prediction is worse than the optimum. This translates to the following definition of smoothness and robustness:

Definition 0.2 (smooth-robust)**.** An algorithm A is said to be β-smooth and α-robust if

$$ALG(\sigma) \leq min\{\beta \cdot (1+\delta), \alpha\} \cdot OPT(\sigma)$$

In other words, the competitive ratio of ALG is at most $\min\{\beta \cdot (1+\delta), \alpha\}$ where δ is the relative error of the prediction. In particular, the competitive ratio of $1+\epsilon$-smooth and α-robust algorithm is at most $\min\{(1+\epsilon) \cdot (1+\delta), \alpha\}$ for all input sequences σ and all possible predictions.

We first trivially observe that for any problem, one cannot be better than 1-smooth. This is since for $\delta = 0$ (i.e. the prediction is completely correct) the algorithm is required to achieve the performance of the optimum and obviously cannot be better than 1-competitive.

[1] There may be multiple optimal algorithms for serving σ, in which case OPT is arbitrarily chosen from them.

Our Results

Our main results for the list update problem with prediction, also known as learning augmented algorithms for list update, are as follows. We differentiate between cases where the prediction is allowed to make paid exchanges and cases where it is not.

- We design two $(1+\epsilon)$-smooth and $O(1/\epsilon^2)$-robust randomized algorithms, assuming the prediction is allowed to make only free exchanges.
- If we allow the prediction to make paid exchanges in addition to free exchanges, we design a $(1+\epsilon)$-smooth and $O(1/\epsilon^4)$-robust randomized algorithm. It is important to note that we also limit the prediction from making a large number of paid exchanges, restricting it to the cost of the optimal solution (up to a constant).

One might wonder about the dependence on ϵ in the robustness if a $(1+\epsilon)$-smooth algorithm is required. We partially address this question by proving the following lower bounds:

- Any deterministic algorithm that is $(1+\epsilon)$-smooth must be at least $\Omega(1/\epsilon)$-robust.
- Any randomized algorithm that is $(1+\epsilon)$-smooth must be at least $\Omega(\log(1/\epsilon))$-robust. The lower bound is established in the "$i-1$" model, where accessing the first element costs 0 and the second one costs 1.

To define our algorithms while allowing the prediction to make paid exchanges, we demonstrate an interesting phenomenon. Any algorithm that accesses an element at a certain location can be converted into another algorithm with at most the same cost, which only changes the order of the elements up to that location of the accessed element and no further. It was a folklore to assume that exchanging two elements, both behind the accessed element, could be postponed. However, accessing an element may necessitate moving a few elements before it to the end. This cannot be entirely postponed as it affects the cost of accessing nearly all elements. Our transformation demonstrates that such swaps should be partially executed without increasing the total cost.

Another crucial component we leverage is the $MTF\epsilon$ algorithm. This algorithm, when requesting an element, moves it to the front with probability ϵ (independent of the history). We observe that this algorithm achieves a competitive ratio of $O(1/\epsilon)$. We utilize this in our algorithms to maintain robustness.

In many online algorithms with predictions, a common approach is to combine an algorithm designed for the prediction with a standard competitive algorithm for the problem. Typically, this is achieved by alternating between the two algorithms over time, introducing an inherent loss of a multiplicative constant (2 at best). However, since the list update problem already has a 2-competitive algorithm, this technique is not as potent. Instead of combining algorithms over time, we propose combining them over space, specifically by integrating the prediction list with, for example, the 2-competitive MTF algorithm's list. For example, a natural algorithm will follow the prediction while interleaving (approximately every $1/\epsilon$ elements in its list) the elements of the list of the MTF. However, maintaining this state is too expensive with respect to MTF elements. Hence, one might be tempted to combine the prediction with the $MTF\epsilon$ algorithm, which makes much fewer changes. However, such an algorithm still fails. Consequently, we need to combine the two lists in a dynamic fashion that is not a direct function of the two lists (prediction and $MTF\epsilon$). This quite non-trivial combination is done without fixed slots and is dependent on the history (and the current state).

Previous Work

We begin by reviewing previous work related to the classical List Update problem. Sleator and Tarjan (Sleator and Tarjan 1985) initiated this research by introducing the deterministic online algorithm Move to Front (MTF). Upon receiving a request for an element e, this algorithm accesses e and then moves e to the beginning of the list. They proved that MTF is 2-competitive in a model where free swaps to the accessed element are allowed. They established a simple lower bound of 2 for the competitive ratio of deterministic online algorithms by always requesting the last element in the online algorithm's list. Randomized upper bounds for the competitive ratio have been explored by various researchers (Irani 1991; Reingold, Westbrook, and Sleator 1994; Albers and Mitzenmacher 1997; Albers 1998; Ambühl, Gärtner, and Stengel 2000), with Albers, Von Stengel, and Werchner (Albers, Von Stengel, and Werchner 1995) achieving a competitive ratio of 1.6 with a random online algorithm. Lower bounds have also been investigated (Teia 1993; Reingold, Westbrook, and Sleator 1994; Ambühl, Gärtner, and Stengel 2000), with Ambühl, Gartner, and Von Stengel (Ambühl, Gärtner, and Von Stengel 2001) establishing a lower bound of 1.50084 on the competitive ratio for the classical problem. Ambühl also proved the NP-hardness of the offline classical problem (Ambühl 2000). Furthermore, List Update with Time Windows or delay function was considered by (Azar, Lewkowicz, and Vainstein 2023), who provided constant competitive algorithms for these more general frameworks.

There is a growing body of work in online algorithms with predictions in recent years (see, e.g., over 180 papers in https://algorithms-with-predictions.github.io/). Lykouris and Vassilvitskii introduced this model for the caching problem (Lykouris and Vassilvitskii 2021), and it has since been applied to various problem classes, including rent or buy (Kumar, Purohit, and Svitkina 2018; Khanafer, Kodialam, and Puttaswamy 2013; Gollapudi and Panigrahi 2019; Wei and Zhang 2020; Anand, Ge, and Panigrahi 2020; Wang, Li, and Wang 2020), scheduling (Kumar, Purohit, and Svitkina 2018; Wei and Zhang 2020; Bamas et al. 2020; Lattanzi et al. 2020; Mitzenmacher 2020; Lee et al. 2021; Azar, Leonardi, and Touitou 2021), caching (Lykouris and Vassilvitskii 2018; Wei 2020; Jiang, Panigrahi, and Sun 2020; Bansal et al. 2020), matching (Lavastida et al. 2021; Dütting et al. 2021; Antoniadis et al. 2020b; Jiang et al. 2021), graph problems (Antoniadis et al. 2020a; Jiang et al. 2022; Almanza et al. 2021; Bernardini et al. 2022; Anand et al. 2022;

Fotakis et al. 2021; Azar, Panigrahi, and Touitou 2022), and more.

The Algorithm

We assume that $\epsilon \leq \frac{1}{2}$, otherwise, just use the algorithm with $\epsilon = \frac{1}{2}$. In addition, we assume that $1/\epsilon$ is an integer, otherwise round it down to the closest appropriate value. Our algorithm uses the $MTF\epsilon$ algorithm. Every time an element is requested, with probability of ϵ (independent of all other randomness in the algorithm), the $MTF\epsilon$ algorithm moves the requested element to the front and with probability $1-\epsilon$ it keeps the requested element in its current position.

The Algorithm and the Invariant

The algorithm we are about to present was defined in advance with the purpose of maintaining a specific invariant. We believe it will be easier to understand the algorithm after the invariant has been presented, thus we present the invariant first. Let E be the set of elements where $|E| = n$. Let x_e to be the position of $e \in E$ in ALGs list; y_e to be the position of e in $PRED$s list (prediction list) and z_e to be the position of e in $MTF\epsilon$'s list. We also define

$$INV_e = \{i \in E : i \text{ is before } e \text{ in } ALG\text{s list and after it in } PREDs \text{ list}\}.$$

Definition 0.3. For each element $e \in E$, we denote the two inequalities:

- Inequality I_e as the inequality $|INV_e| < \epsilon \cdot x_e$.
- Inequality II_e as the inequality $\epsilon \cdot x_e \leq z_e$.

We say that the invariant holds if and only if inequalities I_e and II_e hold for each element e.

Now that we have defined the invariant, we shall define the algorithm. First, we define it and then analyze it assuming $PRED$ does not perform paid swaps. Only then we add components to the algorithm which deal with paid swaps done by $PRED$ and then analyze the modified algorithm.

The algorithm maintains a simulation of the prediction (a deterministic list) and a specific realization of the list of $MTF\epsilon$. These two simulations are independent of what the algorithm is doing. The algorithm's list is based on these two lists, both current and past. We maintain the invariant described above deterministically for all possible coin flips of $MTF\epsilon$ (though the bounds may be different for each possible coin flips). It will be always true (as we prove by the first inequality of the invariant) that the location of each element in the algorithm is not far behind the one in the prediction, specifically by at most a factor of $(1 + 2\epsilon)$. Moreover, the location of each element in the algorithm is not far by a factor of more than $1/\epsilon$ than the location of $MTF\epsilon$ for every possible coin flips. For every new request e the prediction may move e for free towards the front. We follow the prediction in the following way: while the element preceding e in our list is after the e in the prediction, we swap the elements and repeat. In this way, the first inequality holds, but the second inequality may stop holding for the elements between the old position of e to its new one (because their location is increased by 1 in the algorithm and remain the same in $MTF\epsilon$). Note that these phenomena may happen only to "red" positions, which are multiples of $1/\epsilon$. We fix it by swapping them with the preceding element. Next we are performing $MTF\epsilon$. With probability $1-\epsilon$ we do nothing and otherwise inequality II_e may be violated. In that case, we move e to position $1/\epsilon$. That may violate the first inequality to some elements between the previous position of e and the new one. We scan the list from the head and for such element i we swap it with the preceding element.

Algorithm 1 PRED performs free swaps only

Upon a request for the element e:
1. Access the element e.
2. Do the following:
2.1. $PRED$ may decrease the position of e in its list using free swaps.
2.2. **While** the element preceding e in ALGs list is after e in $PRED$'s list:
2.2.1. Swap it with e, moving e towards the beginning of the list.
2.3. **While** there is an element i that satisfies $\epsilon \cdot x_i > z_i$:
2.3.1. Let i be the farthest element such that $\epsilon \cdot x_i > z_i$.
2.3.2. Swap i with the element preceding it in ALGs list.
3. With probability of ϵ do the following:
3.1. $MTF\epsilon$ performs "move-to-front" on e.
3.2. **If** $x_e > \frac{1}{\epsilon}$ **then**:
3.2.1. ALG moves e to position $\frac{1}{\epsilon}$ in its list.
3.3. **While** there is an element i that satisfies $|INV_i| \geq \epsilon \cdot x_i$:
3.3.1. Let i be the closest element in ALGs list that satisfies $|INV_i| \geq \epsilon \cdot x_i$.
3.3.2. Swap i with its preceding element j in ALGs list.

Our plan from now on is as follows. First, we shall prove that the invariant (as presented in Definition 0.3) always holds. Then we use this fact in order to prove the smoothness and robustness of Algorithm 1, which henceforth will be denoted by ALG. Then we consider the case where $PRED$ may be allowed to perform paid swaps.

The following two lemmas will be proven in the Appendix "Proof that the Invariant Always Holds" of the full version, where we also prove that both the while loops terminate in linear time.

Lemma 0.4. *Assume the invariant holds at the beginning of line 2.1. Then, at the end of line 2.3, the invariant will hold again.*

Lemma 0.5. *Assume the invariant holds at the beginning of line 3.1. Then at the end of line 3.3, the invariant will hold again.*

These two lemmas imply that:

Lemma 0.6. *The invariant holds prior to every request for an element.*

Smoothness and Robustness Analysis

The purpose of this subsection is to prove the following theorem. Many of the proofs for the lemmas presented in this section are deferred to the Appendix "Smoothness and Robustness Analysis" in the full version.

Theorem 0.7. *If no paid swaps are done by $PRED$ then we have that ALG is $1 + 6\epsilon$-smooth and $\frac{3}{\epsilon^2}$-robust.*

In order to analyze the smoothness and robustness of ALG, we firstly need to bound the expected number of paid swaps ALG performs during lines 2.3 and 3.3. This is done in the following two lemmas.

Lemma 0.8. *The number of paid swaps ALG performs in the loop of line 2.3 is at most $\epsilon \cdot x'_e$, where x'_e is the position of e in ALGs list at the beginning of line 2.1.*

Lemma 0.9. *The number of paid swaps ALG performs in the loop of line 3.3 is at most x'_e, where x'_e is the position of e in ALGs list at the beginning of line 3.1.*

Before we continue, we give the following definition.

Definition 0.10. For each algorithm A and a request $r \in \sigma$ we denote by $A(r)$ to be the cost ALG pays for serving the request r. In addition, we have $A(\sigma) = \sum_{r \in \sigma} A(r)$.

The following two lemmas are used in order to bound the cost ALG pays for a single request using the cost $PRED$ pays for this request.

Lemma 0.11. *Consider a request r for the element e. Then we have $\mathbb{E}[ALG(r)] \leq (1 + 2 \cdot \epsilon) \cdot x_e$.*

Lemma 0.12. *For an element $e \in E$, if inequality I_e holds then we have $x_e \leq (1 + 2 \cdot \epsilon) \cdot y_e$.*

Now we combine the two lemmas above in order to bound the cost ALG pays in order to serve a request using the cost $PRED$ pays in order to serve that request.

Lemma 0.13. *Consider a request r for the element e. Then we have $\mathbb{E}[ALG(r)] \leq (1 + 6 \cdot \epsilon) \cdot y_e$.*

Now we are ready to show smoothness. Observe that since $PRED$ does not perform paid swaps, the cost it pays for serving a request to the element e is y_e (for accessing this element). Therefore, by using lemma 0.13 and summing for all the requests, we get the following corollary.

Corollary 0.14. *We have $\mathbb{E}[ALG(\sigma)] \leq (1 + 6 \cdot \epsilon) \cdot PRED(\sigma)$.*

Now that we dealt with smoothness, we prove robustness.

Lemma 0.15. *We have $\frac{\mathbb{E}[ALG(\sigma)]}{OPT(\sigma)} \leq \frac{3}{\epsilon^2}$.*

Now we are ready prove Theorem 0.7.

Proof of Theorem 0.7. The theorem follows from corollary 0.14 and lemma 0.15. □

Pre-processing of $PRED$

Until now, we defined and analyzed ALG assuming $PRED$ does not perform any paid swaps. In the upcoming subsections, we deal with the case where $PRED$ may perform paid swaps.

Recall that in the List Update problem, any algorithm (and in particular $PRED$ that we are comparing with) may perform paid swaps at any time between any two consecutive elements in its list. However, it will be convenient for us to assume that $PRED$ being "lazy" in the sense that (a) it may perform paid swaps only prior to accessing an element, (b) if $PRED$ is about to access an element located at position y in its list then it may perform paid swaps only with the prefix of the first y elements in its list and (c) $PRED$ performs these paid swaps in a specific order which we explain later.

In the Appendix "Pre processing of $PRED$" of the full version, we show how any algorithm A for the List Update problem (and in particular, $PRED$) can be converted to another algorithm $LAZY_A$ which acts according to the three constraints above and, in additional, satisfies $LAZY_A(\sigma) \leq A(\sigma)$ for each σ. This means that proving smoothness against $LAZY_{PRED}$ yields also to smoothness against the original $PRED$. Note that the result of Appendix "Pre processing of $PRED$" is a general result for the List Update problem and is interesting by itself. Specifically, it shows that any algorithm can be converted to a lazy one without increasing the cost. We believe such claim was not proved before since it was not required before.

Note that from now on (i.e. in the upcoming subsections) we assume that the pre-processing defined in Appendix "Pre processing of $PRED$" is already done in the background (i.e. we are comparing our algorithm with $LAZY_{PRED}$ rather than with $PRED$). When we refer to $PRED$ in the upcoming subsections, we will actually refer to $LAZY_{PRED}$. Therefore, when proving the smoothness of ALG compared with $PRED$, it means actually proving smoothness compared with $LAZY_{PRED}$ rather than $PRED$. Since $LAZY_A(\sigma) \leq A(\sigma)$, any algorithm which is c-competitive against $LAZY_{PRED}$ is also c-competitive against $PRED$. Therefore, the results which we are about to prove hold for the original prediction as well.

ALG reacts to a single paid swap done by $PRED$

In this subsection we explain how ALG reacts when $PRED$ performs a single paid swap. In the following procedure, we deal with the situation where $PRED$ performs a paid swap between i and j where i was the element preceding j in $PRED$'s list prior to this swap. We define k to be the element preceding j in ALGs list at that time (if exist, i.e. if $x_j > 1$). We define p to be the element preceding k in ALGs list at that time (if exist, i.e. if $x_j > 2$). In other words, the order of the three elements mentioned above in ALGs list when $PRED$ performs this swap is p, k, j (although we may have that p does not exist or we may have that both p and k do not exist).

Observe that if the condition in line 2 holds then we must have $x_j \geq 2$ (i.e. k must exist). If we also have that the condition of line 2.1 does not hold then we must have that there exists an integer r such that

$$\epsilon \cdot (x_j - 1) \leq r < \epsilon \cdot x_j.$$

This implies that $x_j \geq 3$ (i.e. p must exist) and thus Algorithm 2 is well defined.

Algorithm 2 PRED performs a paid swap

1. PRED performs the swap between i and j
2. **If** $|INV_j| \geq \epsilon \cdot x_j$ **then**
2.1. **If** $\lceil \epsilon \cdot (x_j - 1) \rceil = \lceil \epsilon \cdot x_j \rceil$ **then**
2.1.1. Swap between k and j
2.2. **Else**
2.2.1. **If** $\epsilon \cdot (x_k + 1) > z_k$ **then**
2.2.1.1. Re-order the elements as j, k, p
2.2.2. **Else**
2.2.2.1. Re-order the elements as j, p, k

The following lemma will be proven in Appendix "Proof that the Invariant Always Holds".

Lemma 0.16. *Assume the invariant holds prior to PRED performing a paid swap. It will hold again after executing Algorithm 2.*

The following definition applies to the following subsections.

Definition 0.17. In the following sections, when we refer to ALG, we refer to Algorithm 1 with the prepossessing of $PRED$ defined in subsection "Pre-processing of $PRED$" and with the treatment of each paid swap done by the prepossessed prediction as explained in Algorithm 2.

Smoothness and Robustness Analysis in case $PRED$ performs paid swaps

In Appendix "Smoothness and Robustness Analysis in case $PRED$ Performs Paid Swaps" we prove the following theorem:

Theorem 0.18. *If $PRED$ is allowed to perform at most $OPT(\sigma)$ paid swaps then we have that ALG is $1+\epsilon$-smooth and $O(\frac{1}{\epsilon^4}) - robust$.*

Lower Bounds

In this section we show lower bounds for the problem. The proofs are deferred to Appendix "Lower Bounds".

We firstly show that any deterministic algorithm which is guaranteed to have cost $(1+\epsilon)PRED$ must be $\Omega(\frac{1}{\epsilon})$ robust. This implies that the dependence on ϵ in our robustness guarantee, indeed necessary for the smoothness we obtain. We assume that $\epsilon \geq \frac{1}{n}$, otherwise, the algorithm which follows the prediction blindly is $\frac{1}{\epsilon}$-robust and 1-smooth, thus the problem is not interesting in this case.

Lemma 0.19. *For any $\epsilon \geq \frac{1}{n}$ there is no deterministic algorithm which is both $1+\epsilon$ smooth and $\frac{1}{3} \cdot \frac{1}{\epsilon}$ robust.*

For the randomized lower bound, the list will only have 2 elements. The head will have access cost 0 and the tail will have access cost 1 (this is also called the "$i-1$" model). The prediction will never perform any exchanges whatsoever. At each time step, the requested element will be whichever the algorithm is more likely to have at the tail.

We show that any algorithm which is guaranteed to have cost $(1+\epsilon)PRED$ must be $\Omega(\log(\frac{1}{\epsilon}))$ robust. This implies that the dependence on ϵ in our robustness guarantee, indeed necessary for the smoothness we obtain.

Lemma 0.20. *Any algorithm equipped with a prediction that achieves expected cost at most $(1+\epsilon)PRED$ must have worst-case competitive ratio $\Omega(\log(\frac{1}{\epsilon}))$ when the head has access cost 0 even if the prediction does not actually use paid exchanges.*

Experiments

We describe below the input generation, the algorithms we evaluated and the results.

Input Generation. We create sequences on n elements (the default value is $n = 100$ but we also tried $n = 30$ and $n = 50$). The length of the sequence is m requests (the default value is $m = 10000$ but we also tried $m = 1000$ and $m = 3000$). We start with a uniform distribution on the elements. At any time, we have our distribution on the elements, and we sample one request. We repeat sampling for a block of requests of certain length B (where B is between 1 and 100). At the end of the block, we update the distribution by updating the probability of B random elements (and renormalize). We call this element the chosen element. Updating to the chosen element is done either through exponential decay or multiplicative weights. Specifically, in the exponential decay, we multiply the vector of probabilities by $\gamma < 1$ and add $1 - \gamma$ to the chosen element (where γ ranges between 0.9 and 0.99). In the multiplicative weights, we multiply the weight of the chosen element by γ (and renormalize, where γ ranges between 1.01 and 1.25). Overall, we have four parameters: n, m, B, γ.

Baselines and Evaluation. We evaluate our algorithm ALG described against one baseline (proxy for OPT) and 4 other algorithms. The baseline would be the prediction $PRED$ which is *decreasing probability*, which is supposed to be very close to the optimum solution. Then the other 4 algorithms (a) the MTF algorithm : the standard online algorithm without predictions. (b) the $MTF\epsilon$ algorithm. (c) the timestamp algorithm. (d) the bit algorithm. The parameter we have here is ϵ (where ϵ ranges between 0.01 to 0.2). For each choice of the five parameters (four parameters for generating the sequence and one parameter for the algorithms), we measure the costs of ALG, the baseline $PRED$ and the four other algorithms. These costs are then averaged over different random inputs.

Experimental Results. In our experiments, we vary on various parameters: In generating the sequence, we vary on the 4 parameter: n the number of elements, m the length of the sequence, B the size of a block, The updated probability γ. Note that in generation of a sequence with exponential decay $\gamma < 1$ and in the multiplicative weights method $\gamma > 1$.

In our evaluation, we vary on the parameter ϵ of ALG and at the same time the parameter of $MTF\epsilon$. The tables of the results and the graphs appear below. We conclude that our algorithm typically performs better than other classical algorithms for this problem. For completeness, the machine used was an HP Omen 15 with an i7-8750H CPU with 16 GB DDR4-2666 SDRAM. The GPU was a GTX 1060 6GB but was unused for this experiment

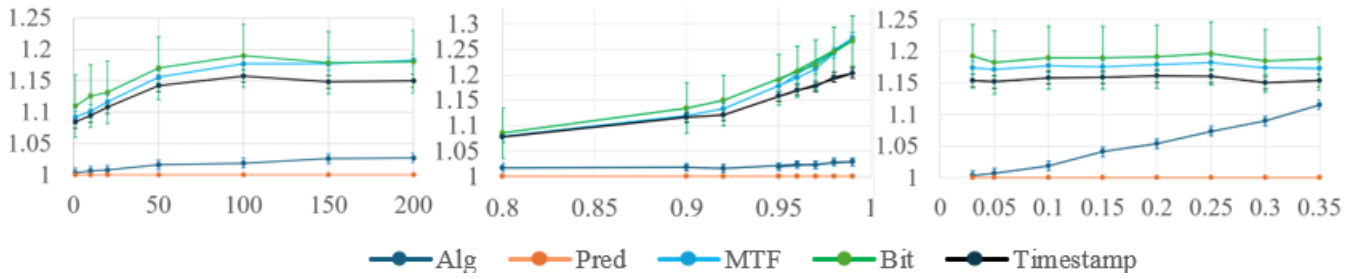

Figure 1: This figure shows the performance of various algorithms relative to the prediction in the exponential decay model; the left plot varies the block size, the center plot the update rate and the right plot epsilon (all on the x-axis) while performance ratio of the methods' cost relative to "follow the prediction" is the y-axis. We omit Random MTF because its performance was generally worse than all the other methods

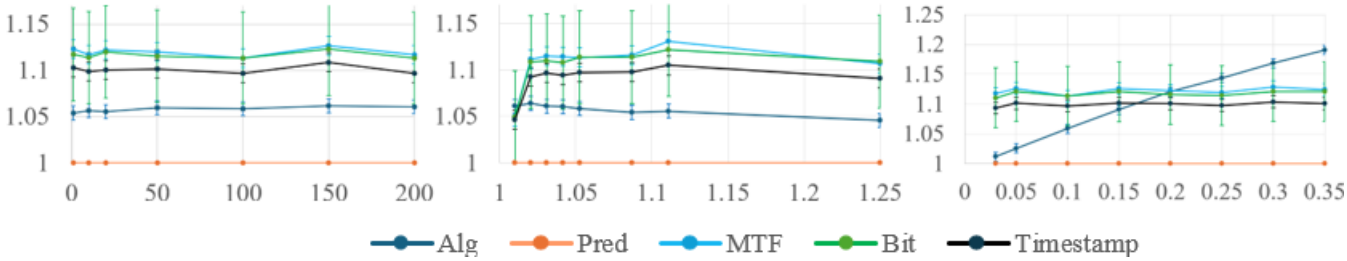

Figure 2: This figure shows the performance of various algorithms relative to the prediction in the multiplicative weights model; the left plot varies the block size, the center plot the update rate and the right plot epsilon (all on the x-axis) while performance ratio of the methods' cost relative to "follow the prediction" is the y-axis. We omit Random MTF because its performance was generally worse than the other methods

Discussion and Open Problems

We considered the list update problem, which is one of the most fundamental problems in online algorithms. Instead of having performance which is far from the optimum by a constant factor, we were interested to have a performance ratio of $1+\epsilon$ assuming we have a correct or close to the correct prediction. We showed that we can actually achieve it (smoothness) while maintaining pretty good worse case performance (robustness) in case the prediction is far from being correct. Our $1+\epsilon$ smooth algorithm is randomized while achieving robustness of $O(1/\epsilon^4)$. Our lower bound for deterministic algorithm is $\Omega(1/\epsilon)$ and only $\Omega(\log 1/\epsilon)$ for randomized ones. In our experiments, we observe that our algorithm typically performs better than other classical algorithms to this problem. There are various fascinating problems left.

- Our algorithm is randomized. Is there a deterministic $1+\epsilon$ smooth and $O(1/\epsilon)$ robust, or at least $O(1/\epsilon^c)$ robust for some constant c ?
- Can the randomized lower bound for the robustness be improved to $\Omega(1/\epsilon)$? Can the robustness of the randomized algorithm be improved to $O(1/\epsilon)$ or even behind that ?
- We encountered a difference in the robustness bound between the case where we allow paid swaps by the prediction and the case where we do not allow. Is there really a difference between these two cases?

Acknowledgments

Yossi Azar was supported in part by the Israel Science Foundation (grant No. 2304/20).

References

Albers, S. 1998. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27(3): 682–693.

Albers, S.; and Mitzenmacher, M. 1997. Revisiting the COUNTER algorithms for list update. *Information processing letters*, 64(3): 155–160.

Albers, S.; Von Stengel, B.; and Werchner, R. 1995. A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters*, 56(3): 135–139.

Almanza, M.; Chierichetti, F.; Lattanzi, S.; Panconesi, A.; and Re, G. 2021. Online Facility Location with Multiple Advice. In Ranzato, M.; Beygelzimer, A.; Dauphin, Y. N.; Liang, P.; and Vaughan, J. W., eds., *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, 4661–4673.

Ambühl, C. 2000. Offline list update is NP-hard. In *European Symposium on Algorithms*, 42–51. Springer.

Ambühl, C.; Gärtner, B.; and Stengel, B. v. 2000. Optimal projective algorithms for the list update problem. In *Interna-*

tional Colloquium on Automata, Languages, and Programming, 305–316. Springer.

Ambühl, C.; Gärtner, B.; and Von Stengel, B. 2001. A new lower bound for the list update problem in the partial cost model. *Theoretical Computer Science*, 268(1): 3–16.

Anand, K.; Ge, R.; Kumar, A.; and Panigrahi, D. 2022. Online Algorithms with Multiple Predictions. In Chaudhuri, K.; Jegelka, S.; Song, L.; Szepesvári, C.; Niu, G.; and Sabato, S., eds., *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, 582–598. PMLR.

Anand, K.; Ge, R.; and Panigrahi, D. 2020. Customizing ML Predictions for Online Algorithms. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, 303–313. PMLR.

Antoniadis, A.; Coester, C.; Elias, M.; Polak, A.; and Simon, B. 2020a. Online metric algorithms with untrusted predictions. In *Proceedings of the 37th International Conference on Machine Learning,ICML 2020*.

Antoniadis, A.; Gouleakis, T.; Kleer, P.; and Kolev, P. 2020b. Secretary and Online Matching Problems with Machine Learned Advice. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M.; and Lin, H., eds., *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Azar, Y.; Leonardi, S.; and Touitou, N. 2021. Flow time scheduling with uncertain processing time. In Khuller, S.; and Williams, V. V., eds., *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, 1070–1080. ACM.

Azar, Y.; Lewkowicz, S.; and Vainstein, D. 2023. List Update with Delays or Time Windows. *CoRR*, abs/2304.06565.

Azar, Y.; Panigrahi, D.; and Touitou, N. 2022. Online Graph Algorithms with Predictions. In Naor, J. S.; and Buchbinder, N., eds., *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, 35–66. SIAM.

Bamas, É.; Maggiori, A.; Rohwedder, L.; and Svensson, O. 2020. Learning Augmented Energy Minimization via Speed Scaling. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M.; and Lin, H., eds., *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Bansal, N.; Coester, C.; Kumar, R.; Purohit, M.; and Vee, E. 2020. Scale-Free Allocation, Amortized Convexity, and Myopic Weighted Paging. *CoRR*, abs/2011.09076.

Bernardini, G.; Lindermayr, A.; Marchetti-Spaccamela, A.; Megow, N.; Stougie, L.; and Sweering, M. 2022. A Universal Error Measure for Input Predictions Applied to Online Graph Problems. In *NeurIPS*.

Dütting, P.; Lattanzi, S.; Leme, R. P.; and Vassilvitskii, S. 2021. Secretaries with Advice. In Biró, P.; Chawla, S.; and Echenique, F., eds., *EC '21: The 22nd ACM Conference on Economics and Computation, Budapest, Hungary, July 18-23, 2021*, 409–429. ACM.

Fotakis, D.; Gergatsouli, E.; Gouleakis, T.; and Patris, N. 2021. Learning Augmented Online Facility Location. *CoRR*, abs/2107.08277.

Gollapudi, S.; and Panigrahi, D. 2019. Online Algorithms for Rent-Or-Buy with Expert Advice. In Chaudhuri, K.; and Salakhutdinov, R., eds., *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, 2319–2327. PMLR.

Irani, S. 1991. Two results on the list update problem. *Information Processing Letters*, 38(6): 301–306.

Jiang, S. H.; Liu, E.; Lyu, Y.; Tang, Z. G.; and Zhang, Y. 2022. Online Facility Location with Predictions. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Jiang, Z.; Lu, P.; Tang, Z. G.; and Zhang, Y. 2021. Online Selection Problems against Constrained Adversary. In Meila, M.; and Zhang, T., eds., *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, 5002–5012. PMLR.

Jiang, Z.; Panigrahi, D.; and Sun, K. 2020. Online Algorithms for Weighted Caching with Predictions. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020*.

Khanafer, A.; Kodialam, M.; and Puttaswamy, K. P. N. 2013. The constrained ski-rental problem and its application to online cloud cost optimization. In *Proceedings of the INFOCOM*, 1492–1500.

Kumar, R.; Purohit, M.; and Svitkina, Z. 2018. Improving Online Algorithms via ML Predictions. In Bengio, S.; Wallach, H. M.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, 9684–9693.

Lattanzi, S.; Lavastida, T.; Moseley, B.; and Vassilvitskii, S. 2020. Online Scheduling via Learned Weights. In Chawla, S., ed., *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, 1859–1877. SIAM.

Lavastida, T.; Moseley, B.; Ravi, R.; and Xu, C. 2021. Learnable and Instance-Robust Predictions for Online Matching, Flows and Load Balancing. In Mutzel, P.; Pagh, R.; and Herman, G., eds., *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, 59:1–59:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Lee, R.; Maghakian, J.; Hajiesmaili, M. H.; Li, J.; Sitaraman, R. K.; and Liu, Z. 2021. Online Peak-Aware Energy Scheduling with Untrusted Advice. In de Meer, H.;

and Meo, M., eds., *e-Energy '21: The Twelfth ACM International Conference on Future Energy Systems, Virtual Event, Torino, Italy, 28 June - 2 July, 2021*, 107–123. ACM.

Lykouris, T.; and Vassilvitskii, S. 2018. Competitive Caching with Machine Learned Advice. In Dy, J. G.; and Krause, A., eds., *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, 3302–3311. PMLR.

Lykouris, T.; and Vassilvitskii, S. 2021. Competitive Caching with Machine Learned Advice. *J. ACM*, 68(4): 24:1–24:25.

Mitzenmacher, M. 2020. Scheduling with Predictions and the Price of Misprediction. In Vidick, T., ed., *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Reingold, N.; Westbrook, J.; and Sleator, D. D. 1994. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1): 15–32.

Sleator, D. D.; and Tarjan, R. E. 1985. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2): 202–208.

Teia, B. 1993. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47(1): 5–9.

Wang, S.; Li, J.; and Wang, S. 2020. Online Algorithms for Multi-shop Ski Rental with Machine Learned Advice. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M.; and Lin, H., eds., *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Wei, A. 2020. Better and Simpler Learning-Augmented Online Caching. In Byrka, J.; and Meka, R., eds., *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020, August 17-19, 2020, Virtual Conference*, volume 176 of *LIPIcs*, 60:1–60:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Wei, A.; and Zhang, F. 2020. Optimal Robustness-Consistency Trade-offs for Learning-Augmented Online Algorithms. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M.; and Lin, H., eds., *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.